POLLACK PERIODICA
An International Journal for Engineering and Information Sciences
DOI: 10.1556/606.2019.14.1.4
Vol. 14, No. 1, pp. 35–46 (2019)
www.akademiai.com

IMPROVING GRAPHICS PROGRAMMING WITH SHADER TESTS

Ádám István SZŰCS*

Department of Computer Algebra, Faculty of Computer Science, Eötvös Loránd University
Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary, e-mail: szaqaei@inf.elte.hu

Received 30 December 2017; accepted 8 June 2018

Abstract: This paper presents an automated model and a project, Arrakis, for finding defects in shading algorithms for graphics rendering and compute workloads. A key challenge in shading algorithm testing is the lack of an oracle that can determine the quality and the output of a custom shading algorithm; this is crucial in graphics workloads because expensive assets are often wasted on solving these problems. A broad solution, Arrakis is developed, which builds on current graphics technology advances in Vulkan, SPIR-V and SPIRV-X by leveraging the standardization with mappings from SPIR-V and C++. Findings show that utilizing the demonstrated technology can improve quality whilst increasing productivity.

Keywords: Software engineering, Computing methodologies, Rasterization, Graphics programming, Shading

1. Introduction

The fastest evolving hardware, the Graphical Processing Unit (GPU) is one of most popular computer components to program [1]. Its general purpose computational power is often multi-teraflops (floating operations per second), [2] furthermore it carries a significant amount of silicon to implement a partially fixed graphics pipeline.

GPU's pipeline is powered by the technologies OpenGL [3], Vulkan [4] and Direct3D [5] Application Programming Interfaces (APIs). The applications vary from User Interface (UI) [6] to rendering of Volumetric Data for Medical Applications

* Corresponding Author

(VDMA) with complex light simulations [7] or real-time games with outdoor and indoor scenes [8]. To harness the power of GPUs one needs to write programs known as ‘shaders’.

Shader programs are often written in OpenGL Shading Language (GLSL) [9] or High-Level Shading Language (HLSL) [10]. These languages provide high level and portable constructs to program the Vertex, Tessellation, Pixel and Fragment stages of the rendering pipeline.

Execution of shaders is emitted in the GPU driver by dispatching API calls to the hardware. OpenGL drivers are complex programs running allocation optimizations with a built-in shader compiler [11]. In modern drivers for D3D12 and Vulkan, the programmer has broad control without a built-in compiler. Graphics compilation is the transformation that turns the shader code into GPU assembly. The shader is often directly compiled into Instruction Set Architecture (ISA) code.

The direct translations often lead to complicated and underperforming toolchains, therefore Khronos Group created a standardized intermediate language for shader compilers [12]. It is a binary intermediate language representing graphics-shader stages and compute kernels for multiple Khronos APIs, including OpenGL and Vulkan [13]. The success is represented by industry wide support including the open-sourcing of the latest Microsoft HLSL compiler [14].

Even though there is an industry-wide support for these GPU programming technologies, quality assurance remains unsolved. Despite there are implementations for unit testing [15], [16] supporting HLSL shaders [17], these technologies are no longer developed and can’t be applied on current problems of graphics and compute shader solutions.

1.1. Contributions

The work is inspired by different projects on graphics programming, including testing of graphics shader compilers [11], high level constructions of rendering systems in modern game engines [18] and current advances in shader compilation [19]. A technique is presented - preliminary engine - Arrakis for testing and executing GLSL programs based on the intermediate representation [20] and Vulkan.

Arrakis changes the methodology of graphics development by combining test-driven development process with a construction of high-level descriptors in compute and graphics. Implementing rendering and compute solutions often started by writing the shaders based on a Single Instruction Multiple Thread (SIMT) programming model, followed by the construction of API calls on the CPU. Given there is no complete debugging and profiling tools for graphics programming, the solution detects bugs in early stages during development of complex rendering and compute scenarios. The solution aids Vulkan by generating C++ equivalence of shaders. Additionally, the developed solution, Arrakis, provides a high-level construction of resources and renders passes extracted as a graph.

The demonstration is a campaign where an example program was developed with the two aforementioned processes targeting a desktop, workstation environment. The implementation is a ray casting project rendering the voxel version of the Stanford Bunny [21] in a Cornell Box [21].

1.2. Key findings

During the development of the project utilizing Arrakis, many caveats have been found in graphics programming. Key findings are as follows:

Most errors are introduced by changing the programming model

During manual development most of the errors have been identified which were due to the change of programming model between the shader and the host. The counter intuitive change from GLSL language to a CPU code often resulted in errors. The shader language constructions, layout qualifiers and explicit cache handling are different from host side code (C++ has been used to implement host code). Due to the complex nature of Vulkan with its' low level handles and logic, this problem was even more challenging to solve [22].

Host and device setup mismatch results in poor quality and slow development times

Implementing rendering algorithms emitting the setup, parameter setting and draw calls from the host side is inevitable. The modern environment in Vulkan, the Validation Layers (VL) can be used, which can report code setups in many scenarios, incorrect descriptor set binding or update [23]. These tools are powerful but they cannot automatically generate the correct API calls for the shader code, pressuring the development time.

Test-driven development of shaders can improve graphics development

In the scenario it has been found that utilizing Arrakis can yield many benefits in contrast to manual programming of compute and graphics programs. It is possible to generate most of the API setups and constructions from shaders without any additional extensions to GLSL. The automatic C++ 'shader' generation helped to develop rendering and compute modules with test-driven development. The campaign resulted in shorter development time and a significant improvement in quality [24].

In summary, the key contributions are:

- An approach to test-driven development in graphics programming to increase quality and overcome manual testing of computer software;
- An implementation as an engine, Arrakis, for testing and semi-automating development of graphics and compute workloads utilizing GLSL shaders in a Vulkan environment;
- A project to render the Volumetric Stanford Bunny (VSB).

The paper is accompanied by a series of posts and a poster detailing the journey to develop the technology [25], [26].

2. Background

In the project the focus is on Vulkan graphics and compute [23], which is the best cross platform, low driver overhead compute and graphics API. The shading language is GLSL version 460 (*Table I*).

Table I

Test bed for development

Processor	AMD Ryzen R7 1800X
Motherboard	Asus Crosshair VI Hero
Memory	G.Skill 16GB TridentZ DDR4
GPU	2x AMD Radeon R9 Nano 4GB
OS	Windows 10 – Creators Fall Update
Driver	Radeon Crimson 17.7
Vulkan	LunarG SDK 1.0.61.1

Vulkan API only accepts SPIR-V [20] as a shading program language, which is low-level and hardly human readable. Building on this fact, it has been generated from GLSL with the standard GLSlang compiler [12].

Connecting to SPIR-V, Arrakis, is built on top of SPIRV-Cross, which can emit transformations on shader code accepting HLSL, GLSL or GLSL-ES [19]. It has an experimental C++ support, which is extended for the project's purposes and utilized to generate compile time entities.

2.1. Output problem

Testing a rendering algorithm or a simple stage of the rendering pipeline is a challenging task. It is often hard to find a ground truth for a given setup of virtual scene known as an oracle [11], (*Fig. 1*). The task of testing in graphics programming is often done by rendering engineers, who are manually adjusting algorithms to output debugging primitives. They simply check intermediate steps of their programs' output by eye. This task is time consuming and can be error prone in a large code base.

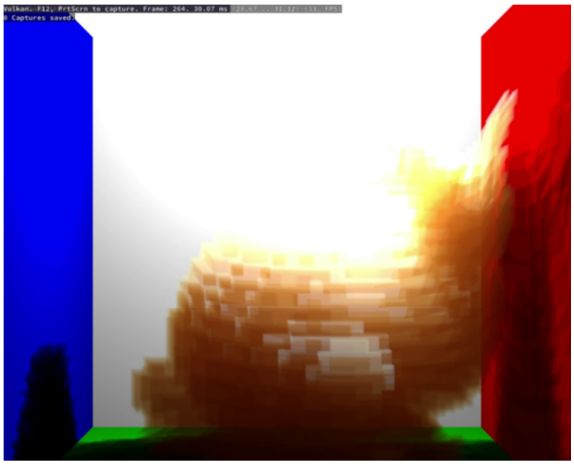

Fig. 1. Rendering error visualizing Volumetric Stanford Bunny in a Cornell Box

Another challenge is, between shader compilers, the results might not be consistent, which complicates the testing of programs [11].

2.2. Test-driven shader development and related work

To overcome the difficulties early in the project life-cycle the process test-driven development is often a viable solution. In Test-Driven Development (TDD) the program is written against the requirements, which are converted into very specific and bounded test cases. During the development the software is improved to pass the tests. This is opposed to the general approach of software development, where modules can be added without any additional tests.

Applying TDD to shader development is not straightforward and the current literature is moderate on addressing problems [22]. Solutions often test the final rendered images with a quality criteria or measure based on image quantities [27]. These solutions are powerful in capturing complex problems in the final rendered image; nevertheless, they are hard to apply early in the development process.

Smart image quality assessment algorithm is developed using a self-organizing map, which can handle random scene elements [24]. This method requires feature vectors for training the algorithm to find errors, which can be challenging in the early stages of the development.

Soft-based CPU Vulkan implementation, the project's goal was to implement a software based Vulkan renderer bringing the API to non-supported systems. Although the project is not finished, it could be a viable future solution to test graphics programs on the CPU. The drawback of the solution is that every fixed function state and extensions affecting the states need to be implemented, which is hard to complete, whereas the hardware execution can lead to errors [28].

Automated testing of graphics shader compilers, a complete and extensive project is introduced by Donaldson et al. [11]. The approach focuses on automated tests of graphics compilers. The solution GLFuzz builds on recent advances in compiler testing, known as fuzzing. It automatically fuzzes shader code by creating equivalent graphics programs and testing against their output automatically. The solution is robust and showed success by finding more than 60 distinct bugs in different OpenGL shader compilers. Their findings show that shader compiler defects are prevalent, and that metamorphic testing provides an effective means for detecting them automatically.

The Arrakis project describes a broad testing solution, which can be applied for graphics programming. Based on SPIR-V mappings to C++ , it has been found that turning shader code into equivalent C++, whilst preserving the semantics is possible.

3. Testing approach

An approach, Arrakis for testing of GLSL shaders with a Vulkan environment is presented (*Fig. 2*). Noting it is applicable to HLSL based on standardized intermediate

representation (SPIR-V). Focusing on a ray casting solution for the Volumetric Stanford bunny in a Cornell box the techniques are physically based [29].

Testing in Arrakis is the following:

- *Equivalent C++ shader generation*. Automatically generate SPIR-V and C++ with the utilization of GLSlang and the extension of SPIRV-Cross from the shader programs;
- *Graphics resource generation*. After the static analysis of the code it maps SPIR-V types to high-level constructs, Buffers or Textures based on Interface Blocks or Descriptor set bindings;
- *FrameGraph setup and execution*. Building on compile time information, the developer can build up FrameGraphs, which can speed up development of graphics programs.

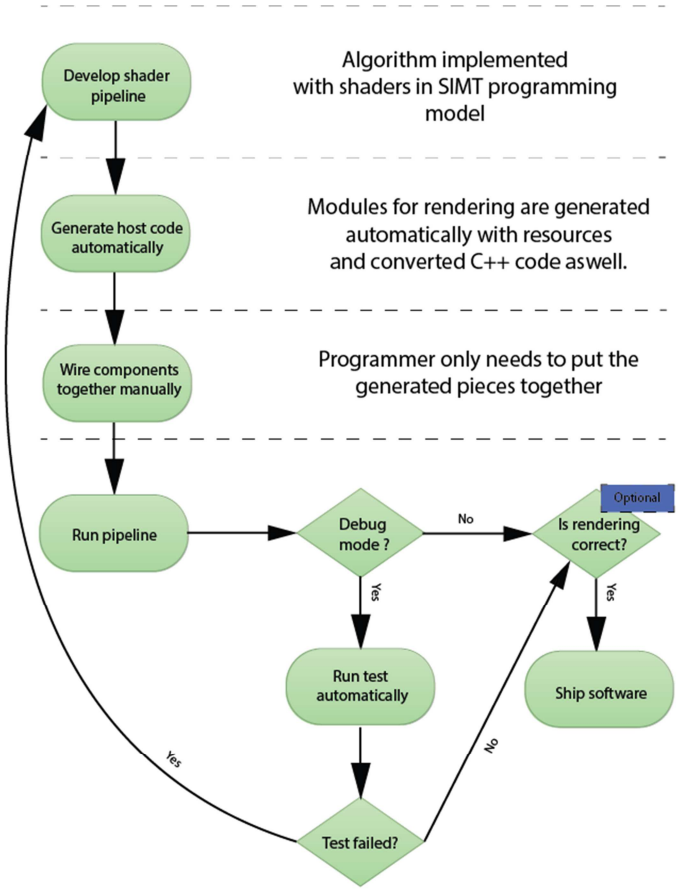

Fig. 2. Steps with Arrakis developing shaders

3.1. C++ shader generation

Equivalent code is generated from the original GLSL shader by mapping the SPIR-V types to standard C++ types with an OpenGL Mathematics (GLM) [30] backend. It must be noted that the floating-point representation of numbers on Graphics Processors are different and modifiable, having a mathematically strict equivalent program is not guaranteed [11].

The solution builds extensively on SPIRV-Cross. The compilation of SPIR-V to C++ works as follows.

Header emission and execution model decision. The first pass of the compilation includes the basic setup of headers and execution modes. The inclusion of headers is easy and straightforward because standard C++ types and interfaces are used only to communicate with the remainder of the renderer. In the next step traversal of the SPIR-V binary with the functionality provided by SPIRV-Cross is performed until a necessary entry point is found to decide execution mode of a shader. This is an important step to complete because the type of the shader helps to determine, which stage special built-in functionality is required. It must be noted that if an execution mode is found, which is not supported an exception is thrown, helping the programmer to find compilation or setup errors in shader code.

Generation of resources. At this stage the solution is equipped with the knowledge of the shader execution type, and the construction of resources for the transformed shader can be started. The list of resources and their mappings are shown in *Table II*. This is a crucial part of the compilation because for testing and setting up of a FrameGraph *Code I* all the necessary types need to be generated and set up.

Emission of shader functions. The next step is the emittance of shader functions. It is a necessary step before emitting the main block because most of the shader code is often organized in functions to keep modularity and readability. Similar to the execution mode decision the solution looks for entries describing user defined functions based on the SPIR-V specification [20]. When a function entry is found the generation of the function body is started with all the local variables and optional control flows. These steps are repeated as long as there are unvisited function entries in the SPIR-V binary. It must be noted; these steps of the generation are only possible because SPIR-V is standard throughout all the vendors and implementations, so the solution can create a mapping between SPIR-V and GLM types.

Table II

Mapping of SPIR-V to Arrakis types

SPIR-V Code	Arrakis Types
OpExecutionMode	ArrakisShaderTransformed
OpTypeImage	ArrakisTexture
OpTypeSampler	ArrakisSampler
OpVariable (Interface, Storage Buffer)	ArrakisBuffer

Code I

Programmer's view of a FrameGraph

```
ArrakisFrameGraph vFrameGraph;

std::vector<ArrakisShaderTransformed*> vCubeShaders;

ArrakisShaderStagev VertexStage=ArrakisShaderStage::VERTEX;
ArrakisShaderStage vFragmentStage=ArrakisShaderStage::FRAGMENT;

vCubeShaders = {new CubeVert(vVertexStage), new CubeFrag(vFragmentStage)};

std::vector<ArrakisShaderTransformed*> vGuiShaders;
vGuiShaders = {new ImguiVert(vVertexStage), new ImguiFrag{vFragmentStage}};

ArrakisImGuiRenderModule::AddRenderGuiPass(vFrameGraph,vCubeShaders);
ArrakisImGuiRenderModule::AddRenderGuiPass(vFrameGraph,vGuiShaders);
```

Emission of decorations and interface override parts. In the last step of the conversion the basic functionality to instantiate a transformed shader is emitted, including

- Transformed shader constructor assembly;
- Stage I/O emittance;
- Interface function overrides.

During this step, finalization of all the setups is completed in the previous stages. At this stage we are assembling the constructor of the transformed shader to set the handles of each of the connecting resources. This is important because in future setups of the FrameGraph, a builder will traverse this list and will omit or occupy a particular resource.

The emission of stage I/O is built in preceding steps and emitted in this part of the solution. The stage inputs and outputs are represented by their transformed Arrakis types and stored as member variables in the generated entities.

After the transformed types and resources are built by Arrakis function overrides are created of a generated shader interface. These cover the main functionality to read, write resources as well as execution of the shader.

After the conversion of shaders from SPIR-V, debugging the transformed graphics programs using a regular unit testing library Google Test or Catch is feasible.

3.2. FrameGraph ordering

During the project of implementing the ray caster with test-driven development obstacles has been found in the development. They were introduced due to the expressive nature of the graphics Vulkan API. During development of the system it was found that the implementation became tied to the rendering API and it became more

limited in extensibility. Due to these shortcomings a FrameGraph architecture was implemented with a Rendering Hardware Interface (RHI) [18].

The FrameGraph is an architecture, which abstracts how render passes and resources are set up and executed during the runtime. The foundation of the system is to represent resources and render passes (A render pass can be seen as a set of draw calls) as nodes and the dependencies between them as edges of a graph can be seen in *Fig. 3*. With the utilization of FrameGraph the system can build up compile time knowledge of a frame before it is rendered or even dispatched for processing. It must be noted to reach the maximum level of modularity and extensibility, this part of Arrakis was strongly built on C++17 features.

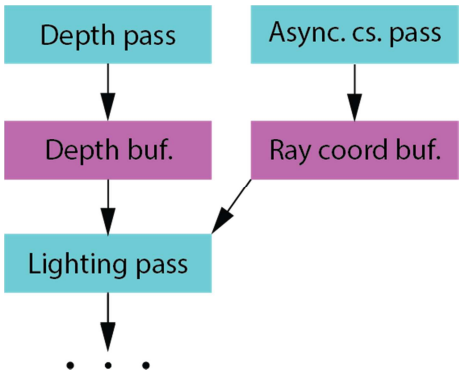

Fig. 3. A visual example of a FrameGraph

Using FrameGraph to implement the solution increased the productivity, while greater control over resources and frame times was reached. A Rendering Hardware Interface was also implemented to decouple the core from third-party headers and implementations. With this extension the solution can be extended to D3D12 also [5], [31].

4. Development time evaluation and measurements

The project's goal was to implement the same ray caster with two methodologies. The first was a manual implementation with no usage of shader transformation or either FrameGraphs, where the latter utilized both of shader transformation and FrameGraph technologies.

For the first approach the functionality with stand-alone classes and functions were implemented. In this scenario the solution was implemented by approximately 5000 Sources Lines of Code (SLOC) excluding the shader programs. During the implementation severe bugs had to be overcome:

- Mismatch between graphics code and CPU sided setup of shader parameters;
- Incorrect allocations in texture and buffer memory;
- Shader layout qualifier (e.g. std450) mismatch affecting byte offset in buffers.

One of the errors is demonstrated in *Fig. 1*, showing a bug introduced by incorrect interpolation in world coordinates. This approach took an experienced programmer almost 3 months to implement the necessary algorithms. During this programming phase all the caveats mentioned in the paper were collected, while developing the necessary technology to overcome these difficulties.

For the second approach both the FrameGraphs with a Rendering Hardware Interface and shader transformations were used. The effective implementation took around 500 SLOC to complete the task excluding the shader codes. For unit testing the graphics code, Google Test was used on the transformed shaders. By using test-driven development for GPU programs, overcoming the aforementioned bugs was possible. The results were significantly better due the high-level control and tests of shader and pipeline code. The entire implementation took 2 - 3 weeks to complete.

5. Concluding remarks and future directions

A large project aimed at test-driven development of shaders was presented with high-level construct over a simple frame. Results show that bugs and errors are frequent due to the expressive nature of modern APIs with no direct control over quality, whereas the change in programming model makes it even harder to solve challenges.

A complex and comprehensive solution, which aims to solve the caveats in graphics programming with mostly compile-time constructs was demonstrated. The solution is broad; however, the Arrakis approach cannot solve all the difficulties in graphics programming. It was found that static generation of converted shaders has significant potential in aiding the programmer during short delivery times. The problem is for large codebases it has caveats, it assumes a well-built continuous integration system and an automated toolchain to compile large number of shaders, whereas handling their extensions is a challenging task.

The plan is to develop Arrakis to handle a broader set of extensions and to implement more scientific applications, as general number systems or partial differential equation solvers. Considering the solution of statically generating resources and modules of the FrameGraph will be kept, however on the execution of shaders a different approach might be taken by implementing a Vulkan layer to lower the toolchain pressure. This approach avoids the need to implement extensions and the execution would be running on the native hardware.

Acknowledgements

This work was supported in part by Advanced Micro Devices Radeon Technologies Group. Information on the Radeon Technologies Group can be obtained from https://radeon.com/. This research was also supported by Zeno Vision Limited and ELTE EIT Digital and by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00001).

Open Access statement

References

[1] Tukora B. Szalay T. High performance computing on graphics processing units, *Pollack Periodica*, Vol. 3, No. 2, 2008, pp. 27–34.
[2] Magoulès F., Ahamed A. K. C., Putanowicz R. Fast iterative solvers for large compressed-sparse row linear systems on graphics processing unit, *Pollack Periodica,* Vol. 10, No.1, 2015, pp. 3–18.
[3] Group K. *OpenGL,* 2017, https://www.khronos.org/registry/OpenGL-Refpages/gl4/, (last visited 1 November 2017).
[4] Group K. *Vulkan,* 2017, https://www.khronos.org/vulkan/, (last visited 1 November 2017).
[5] Microsoft, *Direct3D 12 Programming Guide,* 2017, https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx, (last visited 1 November 2017).
[6] Cornut O. *dearImgui,* https://github.com/ocornut/imgui 2017, (last visited 1 November 2017).
[7] Jönsson D., Kronander J., Ropinski T., Ynnerman A. Historygrams: Enabling interactive global illumination in direct volume rendering using photon mapping, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 18, No. 12, 2012, pp. 2364–2371.
[8] Silvennoinen A., Timonen V. *Multi-scale global illumination in quantum break*, 2015, http://wili.cc/research/quantum_break/, (last visited 13 November 2017).
[9] Group K. *OpenGL shading language,* 2017, https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language, (last visited 1 November 2017).
[10] Microsoft, *HLSL*, 2017, https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx, (last visited 1 November 2017).
[11] Donaldson A. F., Evrard H., Lascu A., Thomson P. Automated testing of graphics shader compilers, *Proc. of the ACM on Program. Lang,* Vol. 1, No. OOPSLA, 2017, Paper No. 93.
[12] Group K. *GlSlang Compiler,* 2017, https://github.com/KhronosGroup/glslang, (last visited 1 November 2017).
[13] Inc. L. *Vulkan,* 2017, https://vulkan.lunarg.com/, (last visited 1 November 2017).
[14] Microsoft, *DirectX Shader Compiler,* 2017, https://github.com/Microsoft/Direct XShaderCompiler, (last visited 1 November 2017).
[15] Google, *GLSL unit is a testing framework in Javascript for WebGL,* 2017, https://code.google.com/archive/p/glsl-unit/, (last visited 1 November 2017).
[16] Rakos D. *Unit testing OpenGL Apple*, 2010, http://rastergrid.com/blog/2010/02/unit-testing-opengl-applications/, (last visited 1 November 2017).
[17] Jones T. *SlimShader,* 2014, https://github.com/tgjones/slimshader, (last visited 1 November 2017).
[18] O'Donnell Y. *FrameGraph: Extensible rendering architecture in frostbite,* 2017, https://www.ea.com/frostbite/news/framegraph-extensible-rendering-architecture-in-frostbite, (last visited 1 November 2017).
[19] Group K. *SPIRV-Cross,* 2017, https://github.com/KhronosGroup/SPIRV-Cross, (last visited 1 November 2017).

[20] Group K. *Standard portable intermediate representation,* 2017, https://www.khronos.org/registry/spir-v/specs/1.2/SPIRV.html, (last visited 1 November 2017).
[21] McGuire M. Computer graphics archive, 2017, http://casual-effects.com/data/index.html, (last visited 1 November 2017).
[22] Lauritzen A. Future directions for compute-for-graphics, 2017, https://www.ea.com/news/seed-siggraph2017-compute-for-graphics, (last visited 14 November 2017).
[23] Sellers G., Kessenich J. *Vulkan programming guide: The official guide to learning Vulkan*, Addison-Wesley, 2016.
[24] Amann J., Weber B., Wüthrich C. A. Using image quality assessment to test rendering algorithms, In: *21st International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision in cooperation with EUROGRAPHICS Association,* M. M. Oliveira, V. Skala (Eds.), Plzen, Czech Republic, 24-27 June 2013, pp. 205–214.
[25] Szucs, A. I. *Unit testing GPU code,* 2017, https://www.hustef.hu/speakerslist/speaker_adamistvanszucs/, (last visited 1 November 2017).
[26] Gralia R. E. *Zeno vision limited,* 2017, https://www.zeno.ai, (last visited 1 November 2017).
[27] Herzog R., Čadík M., Aydčin T. O., Kim K. I., Myszkowski K., Seidel H. P. (2012). NoRM: No-reference image quality metric for realistic image synthesis, *Computer Graphics Forum*, Vol. 31, No. 2, part 3, 2012, pp. 545–554.
[28] Lifshay J. *Vulkan-CPU,* 2017, https://github.com/programmerjake/vulkan-cpu, (last visited 1 November 2017).
[29] Lagarde S., de Rousiers C, *Moving frostbite to physically based rendering 3.0,* Electronic Arts Frostbite, 2014, https://www.slideshare.net/DICEStudio/moving-frostbite-to-physically-based-rendering, (last visited 12 November 2017).
[30] Group K. *OpenGL mathematics*, 2017, https://glm.g-truc.net/0.9.8/index.html, (last visited 1 November 2017).
[31] Games O. *Ashes of the singularity,* 2016, http://www.ashesofthesingularity.com/, (last visited 1 November 2017).